10 | 网络跟踪：如何使用 eBPF 排查网络问题？

2022-02-07 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >

讲述：山荣
时长 17:19 大小 15.87M

你好，我是倪朋飞。

上一讲，我带你一起梳理了应用进程的跟踪方法。根据编程语言的不同，从应用程序二进制文件的跟踪点中可以获取的信息也不同：编译型语言程序中的符号信息可以直接拿来跟踪应用的执行状态，而对于解释型语言和即时编译型语言程序，我们只能从解释器或即时编译器的跟踪点参数中去获取应用的执行状态。

除了前面三讲提到的内核和应用程序的跟踪之外，网络不仅是 eBPF 应用最早的领域，也是目前 eBPF 应用最为广泛的一个领域。随着分布式系统、云计算和云原生应用的普及，网络已经成为了大部分应用最核心的依赖，随之而来的网络问题也是最难排查的问题之一。

那么，eBPF 能不能协助我们更好地排查和定位网络问题呢？我们又该如何利用 eBPF 来排查网络相关的问题呢？今天，我就带你一起具体看看。

eBPF 提供了哪些网络功能？

既然想要使用 eBPF 排查网络问题，我想进入你头脑的第一个问题就是：eBPF 到底提供了哪些网络相关的功能框架呢？

要回答这个问题，首先要理解 Linux 网络协议栈的基本原理。下面是一个简化版的内核协议栈示意图：

极客时间

Linux内核协议栈

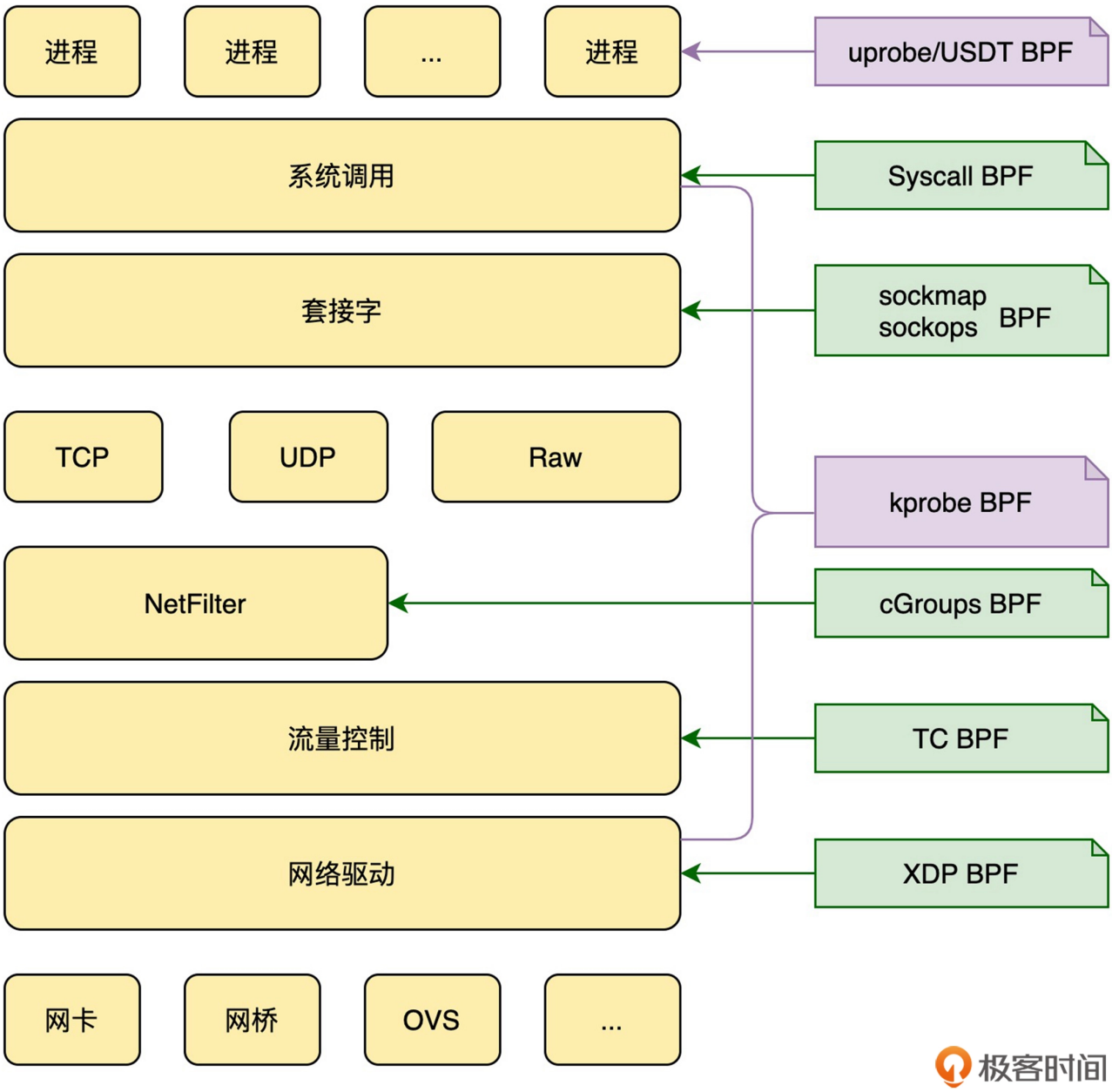
如何理解这个网络栈示意图呢？从上往下看，就是应用程序发送网络包的过程；而反过来，从下往上看，则是网络包接收的过程。比如，从上到下来看这个网络栈，你可以发现：

应用程序会通过系统调用来跟套接字接口进行交互；

套接字的下面，就是内核协议栈中的传输层、网络层和网络接口层，这其中也包含了网络过滤（Netfilter）和流量控制（Traffic Control）框架；

最底层，则是网卡驱动程序以及物理网卡设备。

理解了网络协议栈的基本流程，eBPF 提供的网络功能也就容易理解了。如下图所示，eBPF 实际上提供了贯穿整个网络协议栈的过滤、捕获以及重定向等丰富的网络功能：



内核协议栈 BPF 挂载点

一方面，网络协议栈也是内核的一部分，因而**网络相关的内核函数、跟踪点以及用户程序的函数等，也都可以使用前几讲我们提到的 kprobe、uprobe、USDT 等跟踪类 eBPF 程序进行跟踪**（如上图中紫色部分所示）。

另一方面，回顾一下 06 讲 的内容，**eBPF 提供了大量专用于网络的 eBPF 程序类型，包括 XDP 程序、TC 程序、套接字程序以及 cgroup 程序等。**这些类型的程序涵盖了从网卡（如卸载到硬件网卡中的 XDP 程序）到网卡队列（如 TC 程序）、封装路由（如轻量级隧道程序）、TCP 拥塞控制、套接字（如 sockops 程序）等内核协议栈，再到同属于一个 cgroup 的一组进程的网络过滤和控制，而这些都是内核协议栈的核心组成部分（如上图中绿色部分所示）。

从这两个方面，你可以发现 eBPF 已经涵盖了内核协议栈的很多方面，并且内核社区在网络方面的功能还在不断丰富中。

接下来，我就以最常见的网络丢包为例，带你看看如何使用 eBPF 来排查网络问题。

如何跟踪内核网络协议栈？

即使理解了内核协议栈的基本原理，以及各种类型 eBPF 程序的基本功能，在想要跟踪网络相关的问题时，你可能还是觉得无从下手，这是为什么呢？

究其原因，我认为最主要是因为**不清楚内核中都有哪些函数和跟踪点可以拿来跟踪**。而即使通过源码查询到了一系列的内核函数，还是没有一个清晰的思路把这些内核函数与所碰到的网络问题关联起来。

不过，当你碰到这类困惑的时候不要有畏难心理。要知道，所有非内核开发者都会碰到类似的问题，而解决这类问题也并非难事，下面我们就来看解决方法。

如何把内核函数跟相关的网络问题关联起来呢？看到本小节的标题，你应该已经想到了：跟踪调用栈，**根据调用栈回溯路径，找出导致某个网络事件发生的整个流程，进而就可以再根据这些流程中的内核函数进一步跟踪**。

既然是调用栈的**回溯**，只有我们知道了最接近整个执行逻辑结尾的函数，才有可能开始这个回溯过程。对 Linux 网络丢包问题来说，内核协议栈执行的结尾，当然就是释放最核心的 SKB（Socket Buffer）数据结构。查询内核 SKB 文档，你可以发现，内核中释放 SKB 相关的函数有两个：

第一个，kfree_skb ，它经常在网络异常丢包时调用；

第二个，consume_skb ，它在正常网络连接完成时调用。

这两个函数除了使用场景的不同，其功能和实现流程都是一样的，即都是检查 SKB 的引用计数，当引用计数为 0 时释放其内核内存。所以，要跟踪网络丢包的执行过程，也就可以跟踪 `kfree_skb` 的内核调用栈。

接下来，我就以访问极客时间的网站 time.geekbang.org 为例，来带你一起看看，如何使用 bpftrace 来进行调用栈的跟踪。

为了方便调用栈的跟踪，bpftrace 提供了 kstack 和 ustack 这两个内置变量，分别用于获取内核和进程的调用栈。打开一个终端，执行下面的命令就可以跟踪 kfree_skb 的内核调用栈了：

复制代码

```
sudo bpftrace -e 'kprobe:kfree_skb /comm=="curl"/ {printf("kstack: %s\n", ksta
```

这个命令中的具体内容含义如下：

kprobe:kfree_skb 指定跟踪的内核函数为 kfree_skb；

紧随其后的 /comm=="curl"/ ，表示只跟踪 curl 进程，这是为了过滤掉其他不相关的进程操作；

最后的 printf() 函数就是把内核协议栈打印到终端中。

打开一个新终端，并在终端中执行 curl time.geekbang.org 命令，然后回到第一个终端，就可以看到如下的输出：

复制代码

```
kstack:
        kfree_skb+1
        udpv6_destroy_sock+66
        sk_common_release+34
        udp_lib_close+9
        inet_release+75
        inet6_release+49
        __sock_release+66
        sock_close+21
        __fput+159
        ____fput+14
        task_work_run+103
        exit_to_user_mode_loop+411
        exit_to_user_mode_prepare+187
        syscall_exit_to_user_mode+23
        do_syscall_64+110
        entry_SYSCALL_64_after_hwframe+68

```

```
kstack:
        kfree_skb+1
        udpv6_destroy_sock+66
        sk_common_release+34
        udp_lib_close+9
        inet_release+75
        inet6_release+49
        __sock_release+66
        sock_close+21
        __fput+159
        ____fput+14
        task_work_run+103
        exit_to_user_mode_loop+411
        exit_to_user_mode_prepare+187
        syscall_exit_to_user_mode+23
        do_syscall_64+110
        entry_SYSCALL_64_after_hwframe+68

kstack:
        kfree_skb+1
        unix_release+29
        __sock_release+66
        sock_close+21
        __fput+159
        ____fput+14
        task_work_run+103
        exit_to_user_mode_loop+411
        exit_to_user_mode_prepare+187
        syscall_exit_to_user_mode+23
        do_syscall_64+110
        entry_SYSCALL_64_after_hwframe+68

kstack:
        kfree_skb+1
        __sys_connect_file+95
        __sys_connect+162
        __x64_sys_connect+24
        do_syscall_64+97
        entry_SYSCALL_64_after_hwframe+68

kstack:
        kfree_skb+1
        __sys_connect_file+95
        __sys_connect+162
        __x64_sys_connect+24
        do_syscall_64+97
        entry_SYSCALL_64_after_hwframe+68
```

这个输出包含了多个调用栈，每个调用栈从下往上就是 kfree_skb 被调用过程中的各个函数（函数名后的数字表示调用点相对函数地址的偏移），它们都是从系统调用（entry_SYSCALL_64）开始，通过一系列的内核函数之后，最终调用到了跟踪函数。

输出中包含多个调用栈，是因为同一个内核函数是有可能在多个地方调用的。因此，我们需要对它进一步改进，加上网络信息的过滤，并把源 IP 和目的 IP 等基本信息也打印出来。比如，我们访问一个网址，只需要关心 TCP 协议，而其他协议相关的内核栈就可以忽略掉。

kfree_skb 函数的定义格式如下所示，它包含一个 struct sk_buff 类型的参数，这样我们就可以从中获取协议、源 IP 和目的 IP 等基本信息：

复制代码

```
1 void kfree_skb(struct sk_buff * skb);
```

由于我们需要添加数据结构读取的过程，为了更好的可读性，你可以把这些过程放入一个脚本文件中，通常后缀为 .bt。下面就是一个改进了的跟踪程序：

复制代码

```
1  kprobe:kfree_skb /comm=="curl"/
2  {
3    // 1. 第一个参数是 struct sk_buff
4    $skb = (struct sk_buff *)arg0;
5
6    // 2. 从网络头中获取源IP和目的IP
7    $iph = (struct iphdr *)($skb->head + $skb->network_header);
8    $sip = ntop(AF_INET, $iph->saddr);
9    $dip = ntop(AF_INET, $iph->daddr);
10
11   // 3. 只处理TCP协议
12   if ($iph->protocol == IPPROTO_TCP)
13   {
14     // 4. 打印源IP、目的IP和内核调用栈
15     printf("SKB dropped: %s->%s, kstack: %s\n", $sip, $dip, kstack);
16   }
17 }
```

让我们来看看这个脚本中每一处的具体含义：

第 1 处是把 bpftrace 的内置参数 `arg0` 转换成 SKB 数据结构 `struct sk_buff *`（注意使用指针）。

第 2 处是从 SKB 数据结构中获取网络头之后，再从中拿到源 IP 和目的 IP，最后再调用内置函数 `ntop()` ，把整数型的 IP 数据结构转换为可读性更好的字符串格式。

第 3 处是对网络协议进行了过滤，只保留 TCP 协议。

第 4 处是向终端中打印刚才获取的源 IP 和目的 IP，同时也打印内核调用栈。

直接把这个脚本保存到文件中，bpftrace 并不能直接运行。你会看到如下的类型未知错误：

复制代码

```
./dropwatch.bt:9:10-28: ERROR: Unknown struct/union: 'struct sk_buff'
  $skb = (struct sk_buff *)arg0;
         ~~~~~~~~~~~~~~~~~~
./dropwatch.bt:12:10-26: ERROR: Unknown struct/union: 'struct iphdr'
  $iph = (struct iphdr *)($skb->head + $skb->network_header);
         ~~~~~~~~~~~~~~~~
./dropwatch.bt:13:10-22: ERROR: Unknown identifier: 'AF_INET'
  $sip = ntop(AF_INET, $iph->saddr);
         ~~~~~~~~~~~~
```

这是因为，bpftrace 在将上述脚本编译为 BPF 字节码的过程中，找不到这些类型的定义。在内核支持 BTF 之前，这其实是所有 eBPF 程序开发过程中都会遇到的一个问题。要解决这个问题，我们就需要把所需的内核头文件引入进来。

这里给你一个小提示：v0.9.3 或更新版本的 bpftrace 已经支持 BTF 了，但需要新版本的 libbpf，且还有很多的限制。

那么，如何找出这些数据结构的头文件呢？我通常使用下面的两种方法：

第一种方法是在内核源码目录中，通过查找的方式，找出定义了这些数据结构的头文件（后缀为 `.h`）。

另外一种方法是到 https://elixir.bootlin.com/ 上选择正确的内核版本后，再搜索数据结构的名字。我在下面还会详细介绍这个网站的使用方法。

我们在脚本文件中加入这些类型定义的头文件：

复制代码

```
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/socket.h>
#include <linux/netdevice.h>
```

然后，保存到文件 dropwatch.bt中（你也可以在 GitHub 中找到完整的程序），就可以通过 sudo bpftrace dropwatch.bt 来运行了。

如何排查网络丢包问题？

有了 eBPF 跟踪脚本之后，它能不能用来排查网络丢包问题呢？我们来验证一下。

最常见的丢包是由系统防火墙阻止了相应的 IP 或端口导致的，你可以执行下面的nslookup命令，查询到极客时间的 IP 地址，然后再执行iptables 命令，禁止访问极客时间的 80 端口：

复制代码

```
# 首先查询极客时间的IP
$ nslookup time.geekbang.org
Server:         127.0.0.53
Address:    127.0.0.53#53

Non-authoritative answer:
Name:    time.geekbang.org
Address: 39.106.233.176

# 然后增加防火墙规则阻止80端口
$ sudo iptables -I OUTPUT -d 39.106.233.176/32 -p tcp -m tcp --dport 80 -j DRO
```

防火墙规则加好之后，在终端一中启动跟踪脚本：

复制代码

```
sudo bpftrace dropwatch.bt
```

然后，新建一个终端，访问极客时间，你应该会看到超时的错误：

复制代码

```
$ curl --connect-timeout 1 39.106.233.176
curl: (28) Connection timed out after 1000 milliseconds
```

返回第一个终端，你就可以看到 eBPF 程序已经成功跟踪到了内核丢包的调用栈信息，如下所示：

复制代码

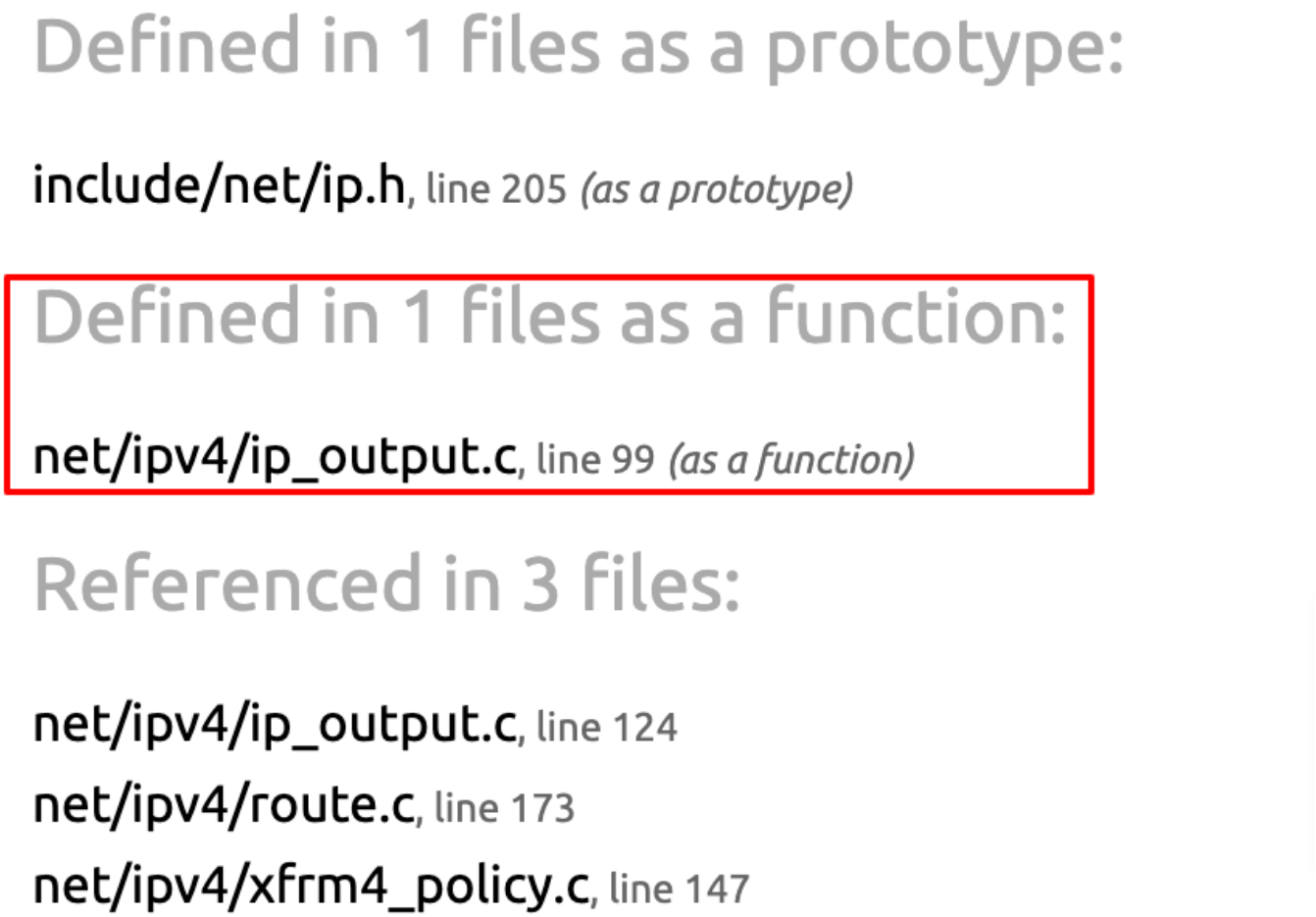
```
SKB dropped: 192.168.1.129->39.106.233.176, kstack:
        kfree_skb+1
        __ip_local_out+219
        ip_local_out+29
        __ip_queue_xmit+367
        ip_queue_xmit+21
        __tcp_transmit_skb+2237
        tcp_connect+1009
        tcp_v4_connect+951
        __inet_stream_connect+206
        inet_stream_connect+59
        __sys_connect_file+95
        __sys_connect+162
        __x64_sys_connect+24
        do_syscall_64+97
        entry_SYSCALL_64_after_hwframe+68
```

从这个输出中，我们可以看到，第一行输出中我们成功拿到了源 IP 和目的 IP，而接下来的每一行中都包含了指令地址、函数名以及函数地址偏移。

从下往上看这个调用栈，最后调用 `kfree_skb` 函数的是 `__ip_local_out`，那么 `__ip_local_out` 这个函数又是干什么的呢？根据函数名，你可以大致猜测出，它是用于向外发送网络包的，但具体的步骤我们就不太确定了。所以，这时候就需要去参考一下内核源代码。

这里推荐你使用 https://elixir.bootlin.com/ 这个网站来查看内核源码，因为它不仅列出了所有内核版本的源代码，还提供了交叉引用的功能。在源码文件中点击任意函数或类型，它就可以自动跳转到其定义和引用的位置。

比如，对于 __ip_local_out 函数的定义和引用，就可以通过 https://elixir.bootlin.com/linux/v5.13/A/ident/__ip_local_out （请注意把 v5.13 替换成你的内核版本）这个网址来查看。点击链接，你会看到如下的界面：



交叉引用搜索结果示意图

查询的结果分为三个部分，分别是头文件中的函数声明、源码文件中的函数定义，以及其他文件的引用。点击中间部分（即上图红框中的第一个链接），就可以跳转到源码的定义位置。

打开代码之后，你会发现，其实并不需要因为不懂内核而担心自己看不懂内核源码，内核中的源码还是很简洁的。这里我把原始代码复制了过来，并且加入了详细的注释，如下所示：

复制代码

```
int __ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
{
    struct iphdr *iph = ip_hdr(skb);

   /* 计算总长度 */
```

```
    iph->tot_len = htons(skb->len);
  /* 计算校验和 */
    ip_send_check(iph);

    /* L3主设备处理 */
    skb = l3mdev_ip_out(sk, skb);
    if (unlikely(!skb))
        return 0;

  /* 设置IP协议 */
    skb->protocol = htons(ETH_P_IP);

  /* 调用NF_INET_LOCAL_OUT钩子 */
    return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT,
                net, sk, skb, NULL, skb_dst(skb)->dev,
                dst_output);
}
```

从这个代码来看，__ip_local_out 函数的主要流程就是计算总长度和校验和，再设置 L3 主设备和协议等属性后，最终调用 nf_hook。

而 nf 就是 netfilter 的缩写，所以你就可以将其理解为调用 iptables 规则。再根据 NF_INET_LOCAL_OUT参数，你就可以知道接下来调用了 OUTPUT 链（chain）的钩子。

知道了发生丢包的问题来源，接下来再去定位 iptables 就比较容易了。在终端中执行下面的 iptables 命令，就可以查询 OUTPUT 链的过滤规则：

复制代码

```
sudo iptables -nvL OUTPUT
```

命令执行后，你应该可以看到类似下面的输出。可以看到，正是我们之前加入的 iptables 规则导致了丢包：

复制代码

```
Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destinatio
    1   180 DROP       tcp  --  *      *       0.0.0.0/0            39.106.233
```

恭喜，到这里，通过简单的几行 bpftrace 脚本，你就成功使用 eBPF 精确定位了一个常见的网络丢包问题。

清楚了问题的根源，要解决它当然就很简单了。只要执行下面的命令，把导致丢包的 iptables 规则删除即可：

```
复制代码
1 sudo iptables -D OUTPUT -d 39.106.233.176/32 -p tcp -m tcp --dport 80 -j DROP
```

如何根据函数偏移快速定位源码？

在内核栈的输出中，我想你一定注意到每一个函数的输出格式都是函数名+偏移量，而这儿的偏移就是调用下一个函数的位置。那么，能不能根据函数名+偏移量直接定位源码的位置呢？

答案是肯定的。这是因为，不仅是我们这些 eBPF 学习者想要这种工具，内核开发者为了方便问题的排查，也经常需要根据内核栈，快速定位导致问题发生的代码位置。所以，Linux 内核维护了一个 faddr2line 脚本，根据函数名+偏移量输出源码文件名和行号。你可以点击这里，把它下载到本地，然后执行下面的命令加上可执行权限：

```
复制代码
1 chmod +x faddr2line
```

在使用这个脚本之前，你还需要注意两个前提条件：

> 第一，带有调试信息的内核文件，一般名字为 vmlinux（注意，/boot 目录下面的 vmlinz 是压缩后的内核，不可以直接拿来使用）。
>
> 第二，系统中需要安装 awk、readelf、addr2line、size、nm 等命令。

对于第二个条件，这些命令都包含在 binutils 软件包中，只需要执行 apt 或者 dnf 命令安装即可。

而对第一个条件中的内核调试信息，各个主要的发行版也都提供了相应的软件仓库，你可以根据文档进行安装。比如，对于 Ubuntu 来说，你可以执行下面的命令安装调试信息：

复制代码

```
codename=$(lsb_release -cs)
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ ${codename}      main restricted universe multive
deb http://ddebs.ubuntu.com/ ${codename}-updates  main restricted universe mul
EOF
sudo apt-get install -y ubuntu-dbgsym-keyring
sudo apt-get update
sudo apt-get install -y linux-image-$(uname -r)-dbgsym
```

而对于 RHEL8 等系统，则可以执行下面的命令：

复制代码

```
sudo debuginfo-install kernel-$(uname -r)
```

调试信息安装好之后，相关的调试文件会放到 /usr/lib/debug 目录下。不同发行版的目录结构是不同的，但你可以使用下面的命令来搜索 vmlinux 开头的文件：

复制代码

```
find /usr/lib/debug/ -name 'vmlinux*'
```

以我使用的 Ubuntu 21.10 为例，查找到的文件路径为 /usr/lib/debug/boot/vmlinux-5.13.0-22-generic。所以，接下来，就可以执行下面的命令，对刚才内核栈中的 __ip_local_out+219 进行定位：

复制代码

```
faddr2line /usr/lib/debug/boot/vmlinux-5.13.0-22-generic __ip_local_out+219
```

命令执行后，可以得到下面的输出：

复制代码

```
__ip_local_out+219/0x150:
```

```
2 nf_hook at include/linux/netfilter.h:256
3 (inlined by) __ip_local_out at net/ipv4/ip_output.c:115
```

这个输出中的具体内容含义如下：

第二行表示 nf_hook的定义位置在 netfilter.h 的156行。

第三行表示 net/ipv4/ip_output.c 的 115行调用了 kfree_skb 函数。但是，由于 nf_hook 是一个内联函数，所以行号115实际上是内联函数 nf_hook 的调用位置。

对比一下上一个模块查找的内核源码，net/ipv4/ip_output.c 的 115 号也刚好是调用 nf_hook 的位置：

```
99  int __ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
100 {
101         struct iphdr *iph = ip_hdr(skb);
102
103         iph->tot_len = htons(skb->len);
104         ip_send_check(iph);
105
106         /* if egress device is enslaved to an L3 master device pass the
107          * skb to its handler for processing
108          */
109         skb = l3mdev_ip_out(sk, skb);
110         if (unlikely(!skb))
111                 return 0;
112
113         skb->protocol = htons(ETH_P_IP);
114
115         return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT,
116                        net, sk, skb, NULL, skb_dst(skb)->dev,
117                        dst_output);
118 }
```

__ip_local_out 源码

而再点击 nf_hook 继续去看它的定义，你可以发现，这的确是个内联函数：

复制代码

```
1 static inline int nf_hook(...)
```

> 小提示：内联是一种常用的编程优化技术，它告诉编译器把指定函数展开之后再进行编译，这样就省去了函数调用的开销。对频繁调用的小函数来说，这就可以大大提高程序的运行效率。

有了 faddr2line 工具，在以后排查内核协议栈时，你就可以根据栈中函数名+偏移量，直接定位到源代码的位置。这样，你就可以直接去内核源码或 elixir.bootlin.com 网站中查找相关函数的实现逻辑，进而更深层次地理解内核的实现原理。

小结

今天，我带你一起梳理了 eBPF 的网络功能，并以最常见的网络丢包问题为例，使用 bpftrace 开发了一个跟踪内核网络协议栈的 eBPF 程序。

eBPF 不仅诞生于网络过滤，它在网络方面的应用也是最为广泛和活跃的。由于内核协议栈也是内核的核心组成部分，前几讲我们讲到的 kprobe 跟踪、uprobe/USDT 跟踪等，也都可以应用到网络问题的跟踪和排查上来。由于内核协议栈相对比较复杂，在排查网络问题时，我们可以从内核调用栈入手。根据调用栈的执行过程，再配合 faddr2line 这样的工具，你就可以快速定位到问题发生所在的内核源码，进而找出问题的根源。

实际上，我们今天讲到的调用栈跟踪也可以用到其他内核功能和用户应用的跟踪上，并且也特别适用于性能优化领域经常需要的热点函数跟踪。在后续的课程中，我还将为你介绍更多的应用案例。

思考题

由于加入了进程信息和网络协议的限制，今天我们使用 bpftrace 开发的 eBPF 程序，实际上只能跟踪到 `curl`命令发出的 TCP 请求丢包问题。而在实际的应用中，很可能是其他的进程发生了丢包问题，并且丢包的也不一定都是 TCP 协议。

那么，根据这一讲的内容和 bpftrace 的文档，你可以对今天的跟踪程序进行改进，并把进程信息（如 PID 和进程名）加到输出中吗？欢迎在评论区和我分享你的思路和解决方法。

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

生成海报并分享

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 用户态跟踪：如何使用 eBPF 排查应用程序？

下一篇 11 | 容器安全：如何使用 eBPF 增强容器安全？

精选留言 (2)

写留言

莫名

2022-02-07

1、比较赞同【最主要是因为不清楚内核中都有哪些函数和跟踪点可以拿来跟踪】这一观点。如果对内核不熟悉，借助 tracepoint 通配符可以很好的协助理解网络协议栈的关键路径。举两个简单例子。

> 例 1：追踪 net 相关追踪点以及调用堆栈：...

展开

作者回复: 非常👍的答案！也谢谢分享跟踪点学习的经验！

5

湛.bin

2022-02-07

ubuntu 18.04 ,snap安装的bpftrace,kernel symbol解析失败
➜ ~ bpftrace -V
bpftrace v0.13.0
➜ ~ echo $BPFTRACE_VMLINUX
/usr/lib/debug/boot/vmlinux-5.4.0-96-generic...

展开

作者回复: 有没有使用snap install --devmode来安装？Github上面也有个类似的issuehttps://github.com/iovisor/bpftrace/issues/1488。旧的发行版最好还是从源码安装 BCC 和 bpftrace。